Ameen Mahouch
CSE 3320 – Operating Systems
Trevor Bakker
29 March 2023

Star Catalog Assignment

Overview

In this assignment, I was given code that calculates the average angular distance between 30,000 stars in the Tycho Star Catalogue. Initially, the given code runs serially, and each preceding star distance must be calculated before the next set of stars calculation executes. With a large number of stars like 30,000, the code takes a significant amount of time to run (precisely $30{,}000^2$ iterations).

The goal of this assignment is to refactor the code to support multithreading, ultimately allowing the program to execute in parallel and optimizing time efficiency. The following was the approach used to multithread:

- Familiarize myself with the given code, including `main.c`, `star.h`, and `utility.h`
- Find a general formula to calculate the start and end indices within the `determineAverageAngularDistance()`function given the number of threads, allowing each thread to have equal amounts of work in parallel.
- Implement `-t` functionality to specify any number of threads.
- Create a separate function for threads to execute, which will be passed in `pthread_create()` along with a struct containing start/end indices for each thread.
- Declare the mean and count variables as global variables and use a mutex lock/unlock to prevent any race conditions.

Libraries

The `pthread.h` library was included to support multithreading. The following were used from the library:

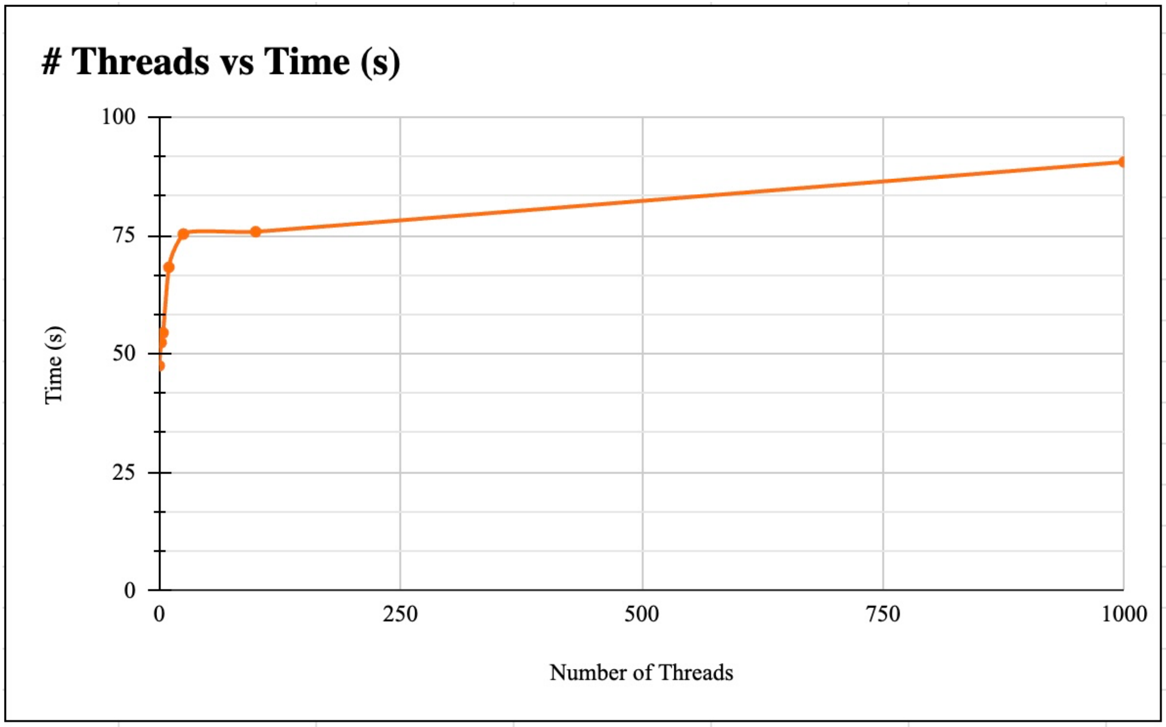
- `pthread_create()` to create a new thread with a start routine and arguments
- `pthread_join()` to suspend execution of a thread until the thread terminates
- `PTHREAD_MUTEX_INITIALIZER` macro to initialize a static mutex
- `pthread_mutex_lock()` to lock the mutex object
- `pthread_mutex_unlock()` to unlock the mutex object

Timing

Instead of implementing code to obtain time values, I thought it was more valuable to utilize the Linux `time` command. In a Bash shell, the time command measures execution time of the executable and outputs real, user, and sys times upon competition. I prefer this method because it does not require any extra code, while also giving time to the microsecond. I executed it as:

```
> make
> time ./findAngular -t 5
```

Results



# Threads	Time (s)	Average Distance	Minimum Distance	Maximum Distance
0	47.477	31.904232	0.000225	179.56972
2	52.366	31.904231	0.000225	179.56972
4	54.429	31.904232	0.000225	179.56972
10	68.278	31.904231	0.000225	179.56972
25	75.286	31.904232	0.000225	179.56972
100	75.776	31.904232	0.000225	179.56972
1000	90.516	31.904231	0.000225	179.56972

Anomalies

During the development of this assignment, there were several abnormalities that appeared.

- The most visible anomaly is how the Average Distance value varies by +/- .000001 after each execution of the program. The likely culprit is floating point resolution in the calculation of the mean. When the mean is calculated, it is divided by a count variable, which can lead to rounding errors that accumulate over time.
- The other large anomaly in this program is how there is no optimal number of threads. An increasing number of threads leads to a logarithmic increase in time. This can most likely be attributed to too much overhead, meaning the amount of overhead in creating/managing threads can offset the potential time efficiency. In this case, the time significantly increases with a mutex lock in place.

- Comparing the baseline performance with the threaded approach, I printed out every single variable for debugging. Most of the variables are the same, except for the count variable. In the baseline approach with no threads, the count variable is equal to 449985000, while the multithreaded version's count terminates around 449985020, having an increase around 20. It is unclear where the inconsistency in the count variable comes as it is locked between a mutex, however there should be no problem as the average distance variable is the same for each approach.

Conclusion

I must note that there were very rare instances when I ran the code locally with 4 threads and achieved a real time of around 46.500 seconds. However, this was rare and achieving this time was rare and inconsistent. The data reflected in my code reflects that there is no optimal thread, however I am convinced through several office hours and class discussion that my data is attributed to Codespace problems. The cloud is through Azure, which could be the source of the problem. The multithreading is done properly with mutexes guarding potentially unsafe variables.

To summarize, *the theoretical optimal number of threads from the code is 4, however it is nondeterministic per each run and rarely achieved*. The averages, the minimum and the maximum values are the same per each run regardless of the number of the threads. This assignment was very interesting and fulfilling as a young developer, thank you!